Automated Grading Assistance For Student Programs

David G. Kay
Department of Information and Computer Science
University of California
Irvine, California 92717
kay@ics.uci.edu

Peter Isaacson
Department of Mathematical Sciences
University of Northern Colorado
Greeley, CO 80639
isaacson@dijkstra.UnivNorthCo.edu

Terry Scott
Department of Mathematical Sciences
University of Northern Colorado
Greeley, CO 80639
tscott@dijkstra.UnivNorthCo.edu

Kenneth A. Reek
Department of Computer Science
Rochester Institute of Technology
One Lomb Memorial Drive
Rochester, NY 14623
kar@cs.rit.edu

Overview: Students are ill-served when grading fails to detect errors in their solutions to programming assignments; they will assume that their development process was adequate when in fact it was not. Yet assignments even in introductory courses can be too complex to evaluate thoroughly by conventional, manual means.

Some instructors have used various software tools to assist in the task of evaluating student programming assignments. The ability to collect students' work in a consistent form and test it automatically against a wide range of data can enhance students' skills in program design, coding, analysis, and testing, and can free the instructor from much tedium, allowing more time for more rewarding activities.

But the use of such tools is far from widespread, despite the impossibility of thorough, valid, reliable assessment without them. No repository of such systems exists, nor do such systems themselves exist for all popular instructional computing platforms.

Automating the testing process is trivial in theory ("for each student, for each test case, produce the result"), but difficult in application. Practically usable systems must be reasonably secure from student tampering, scalable to large classes, flexible enough to accommodate different forms of assignments, and robust in the face of bugs in the students' programs and in the lab environment.

We will share our experiences with such systems, both in their development on various platforms and in their use on students' assignments.

We invite the audience to discuss the implications of these systems, share their own experiences, suggest solutions to still-open issues, and help us begin to compile a catalog of the systems that are available and assess interest in developing, enhancing, and widely disseminating them.

Inadequacy of conventional evaluation: Student programs, even in a first programming course, are typically too complex for an instructor to assess their correctness accurately and comprehensively simply by reading the source code.

The instructor must test the students' programs on various sets of inputs, but the time involved to run each test on each student's program, one by one, is prohibitive. Students can be required to design and run their own tests, submitting the results for grading along with the program's text. However, creating comprehensive, objective, valid sets of test data is a separate skill we also must teach, and thus one which students cannot be relied upon to have mastered. Students' failings in test design thus could detrimentally affect the separate issue of their programs' correctness.

Further, modern computer systems allow any student to "doctor" the results of his or her program to

conceal its errors and omissions; unfortunately this is a temptation that students under deadline or grade pressure sometimes fail to resist.

Automated grading assistance: These tools must recognize that students will misinterpret instructions, will miss deadlines, will fill up file systems, and will introduce all manner of unanticipated errors, both due to misconceptions on their part and inevitable ambiguities in the problem specification. Nonetheless, students must be rewarded to the extent that their efforts are successful. Furthermore, such a system must scale up easily to longer assignments in larger classes. The system should consist of a submittal component and a testing component, as described below.

Submittal component: When a student has completed an assignment, he or she will run the submittal program; this will take a copy of the student's source code and save it somewhere accessible only to the instructor, noting the date and time of submittal.

An ideal submittal system will allow for multiple source files, and for ancillary files of various types: documentation, student-supplied test cases, and so on. It will allow students to resubmit all or part of their assignment, at least up to the due date and perhaps thereafter, keeping track of the new submission time. (Experience shows that it's preferable to accept late assignments appropriately timestamped than to refuse them, because accepting them preserves the state of the student's progress at that moment and the lateness may be renegotiated based on the student's excuse.) The system should also keep a log of each transaction, to support or refute student claims of system failure or unavailability (due, most often, to file systems filling up just before the due date).

When the student submits a program, the system should compile it and even run it against some published test cases. This alerts the student to any gross failures (such as having added a last-minute comment without an ending delimiter) and any unexpected discrepancies with the automated running process. Indeed, one way to use the system is to require the students to run their own programs successfully against a series of tests.

Testing component: After the students have submitted their programs, the testing component takes each program, compiles it, and runs it against every test case that the instructor has specified. It collects the results of every test for each student, printing or displaying them for the instructor to compare with the expected correct results. This approach also promotes the course's curricular goals directly, because it uncovers more flaws in the student's program than conventional grading methods do, which brings home the importance of careful, considered design and testing.

We emphasize, however, that this testing tool does not necessarily compute scores or grades, nor does it reduce the human judgement involved in evaluating students' work (which includes not only the program's correctness, which quality we address here, but also its adherence to the principles of good design, its documentation, and perhaps its user interface or the student's own choice of test data). This tool does provide the evaluator with the comprehensive information necessary to produce an informed, thorough, and fair assessment of correctness. Nor would one generally use automatic comparison tools to match each case's result with the expected one, because too often students misspell words or miscalculate white space, leading to a "false negative" result. For problems of any significance, experience shows that even the most sophisticated case-converters and white-space filters fail to discriminate between significant and trivial errors as well as a human reader's quick perusal can.

This approach also allows the instructor to use automatic tools to examine students' programs for signs of plagiarism; such examination by manual means is sporadic, haphazard, and largely ineffective. Experience with similar systems indicates that after one or two terms of use, when plagiarists have been detected and disciplined, students come to know that such a strategy does not bear fruit and the overall incidence of plagiarism decreases.

Given the complexity of software, even in introductory students' assignments, using this kind of tool is essential. The time has come to devote effort and resources to developing robust, flexible, widely available tools for automated program evaluation. We can no longer pretend to be able to assess students' work validly and reliably by manual means.